Algorithmen und Datenstrukturen 1

SoSe 2020 · Prof. Dr. U. Meyer
27. Juli 2020
Klausurnummer: **0001**

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

Nachname (Druckschrift)**:** _____

Vorname (Druckschrift)**:** _____

Matrikelnummer: _____

Studiengang: _____

Bitte Hinweise beachten:

- Schreiben Sie Ihren **Namen nur auf dieses Titelblatt**. Sollten Sie Zusatzpapier bekommen, vermerken Sie darauf unbedingt die Klausurnummer **0001**.
- Diese Klausur ist für die Variante **Algo1 (vollständig)**.
- Die Klausur besteht aus den Aufgaben **1 − 8**.
- Merken oder notieren Sie sich Ihre Klausurnummer **0001**, da nur unter dieser Nummer die Ergebnisse veröffentlicht werden.
- Es dürfen nur **dokumentenechte Stifte** in den Farben blau und schwarz verwendet werden. Insb. ist die Nutzung von Tintenlöschern untersagt. Zugelassene Hilfsmittel: 1 Blatt DIN A4 mit handschriftlichen Notizen (zweiseitig).
- Das Mitbringen nicht zugelassener Hilfsmittel stellt einen Täuschungsversuch dar und führt zum Nichtbestehen der Klausur. **Schalten Sie bitte deshalb alle elektronischen Geräte, insbesondere Handys und Smartwatches, vor Beginn der Klausur aus und packen Sie diese weg.**
- Werden zu einer Aufgabe zwei oder mehr Lösungen angegeben, so gilt die Aufgabe als nicht gelöst. Entscheiden Sie sich also immer für **eine** Lösung. Begründungen sind nur dann notwendig, wenn die Aufgabenformulierung dies verlangt.
- Die Klausur ist mit Sicherheit bestanden, wenn (ohne Bonifikation aus den Übungspunkten) mindestens **50%** der Höchstpunktzahl erreicht wird.
- Die Klausur dauert **180 Minuten**.

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

Diese Seite ist für den internen Gebrauch.
Bitte leer lassen.

Aufgabe	1	2	3	4	5	6	7	8
Erreichbar	20	22	18	20	20	24	12	24
Erreicht								

Klausur	Bonfikation

2011000100018

Aufgabe 1: Doppelt-verkettete Liste — $2 + 2 + 2 + 14 = 20$ Punkt(e)

In der Vorlesung implementierten wir eine doppelt-verkette Liste zum Speichern von Ganzzahlen (`int`) sinngemäß wie folgt:

```
struct Element {
  int value;
  Element* prev;
  Element* next;
};
```

```
struct List {
  Element* head;
  Element* tail;
};
```

Hier beschreibt `Element*` einen Zeiger (Pointer) auf ein Objekt vom Typ `Element`. Ein Zeiger, der ins „Nichts“ zeigt, hat den Wert `nullptr`. Alle Zeiger haben initial den Wert `nullptr`.

a) Betrachten Sie folgende Funktionen `size` und `is_empty`:

```
// Zähle Elemente in lst
int size(List* lst) {
  Element* it = lst->head;
  int k = 0;
  while(it != nullptr) {
    it = it->next;
    k = k + 1;
  }
  return k;
}
```

```
// Ermittle ob lst leer ist
bool is_empty(List* lst) {
  if (size(lst) == 0) {
    return true;
  } else {
    return false;
  }
}
```

Ermitteln Sie die Laufzeit von `is_empty` asymptotisch exakt[1] und definieren Sie alle verwendeten Variablen.

Laufzeit von `is_empty`: $\Theta(n)$, wobei n die Anzahl der Einträge in `lst` sei

⇑⇑⇑ _____ / 2 Punkt(e) ⇑⇑⇑

b) Implementieren Sie `is_empty_opt` mit best-möglicher Laufzeit in **Pseudocode**.

```
bool is_empty_opt(List* lst) {
  return (lst->head == nullptr);
}
```

⇑⇑⇑ _____ / 2 Punkt(e) ⇑⇑⇑

c) Ermitteln Sie die Laufzeit von `is_empty_opt` asymptotisch exakt und definieren Sie alle verwendeten Variablen.

Laufzeit von `is_empty_opt`: $\Theta(1)$

⇑⇑⇑ _____ / 2 Punkt(e) ⇑⇑⇑

[1] Übereinstimmende obere und untere Schranken.

d) Zeigen Sie, dass wir mit Hilfe von `List` einen **Stack** (Stapelspeicher) simulieren können.

Implementieren Sie hierfür ein effizientes Funktionspaar `push`/`pop` in **Pseudocode** durch explizites Setzen von Zeigern. Analysieren Sie die Laufzeit Ihrer Implementierungen.

```
void push(List* lst, int value) {
  // Gleiches Verhalten wie push auf einem Stack.
  Element* x = new Element();
  x->value = value;
  x->next = lst->head;
  if (lst->tail == nullptr)
    lst->tail = x;
  else
    lst->head->prev = x;
  lst->head = x;









}
```

Laufzeit von `push`: $\Theta(1)$

```
int pop(List* lst) {
  // Gleiches Verhalten wie pop auf einem Stack.
  // Gibt den Wert des entfernten Elements zurück.
  // Sie können annehmen, dass die Liste nicht leer ist.
  int value = lst->head->value; // für späteres return

  if (lst->head == lst->tail)
    lst->tail = nullptr;
  else
    lst->head->next->prev = nullptr;
  lst->head = lst->head->next;

  return value;









}
```

Laufzeit von `pop`: $\Theta(1)$

⇑⇑⇑ _____ / 14 Punkt(e) ⇑⇑⇑

Aufgabe 2: O-Notation und Codeanalyse — $8 + 8 + 6 = 22$ Punkt(e)

a) Geben Sie jeweils eine Funktion f an, welche die angegebenen Wachstumseinschränkungen einhält.

$f \in \omega(\sqrt{n})$	$f \in o(n)$	$f(n) =$ $\sqrt[3]{n^2}$
$f \in \omega(n)$	$f \in o(n \log n)$	$f(n) =$ $n \log \log n$
$4^f \in \Theta(2^f)$		$f(n) =$ 1
$f \in \Theta(n)$	$f \in o(2^n)$	$f(n) =$ $\frac{1}{2}n$

⇑⇑⇑ _____ / 8 Punkt(e) ⇑⇑⇑

b) Lösen Sie die folgenden Rekursionsgleichungen auf. Für alle Rekursionsgleichungen gilt $T(n) = 1$ für alle $n \leq 1$. Nehmen Sie vereinfachend an, dass n so gewählt wurde, so dass sämtliche Divisionen restfrei aufgehen.

$T(n) = 8 \cdot T(\frac{n}{2}) + 3$	$T(n) = \Theta($ n^3 $)$
$T(n) = 2 \cdot T(\frac{n}{4}) + 3$	$T(n) = \Theta($ $\sqrt{n}$ $)$
$T(n) = 1 \cdot T(\frac{n}{2}) + 3$	$T(n) = \Theta($ $\log n$ $)$
$T(n) = 9 \cdot T(\frac{n}{3}) + n^2$	$T(n) = \Theta($ $n^2 \log n$ $)$

⇑⇑⇑ _____ / 8 Punkt(e) ⇑⇑⇑

c) Geben Sie für die folgenden Algorithmen jeweils die Laufzeit in Θ-Notation an.

Algorithmus loop1(n)
  for $i = 1$ **to** n
    $j = i$
    while $j > 1$
      $j = \lfloor j/2 \rfloor$

Algorithmus loop2(n)
  $s = 2$
  while $s \leq n$
    $s = s * s$

Algorithmus loop3(n)
  $i = n$
  while $i \geq 1$
    $j = i$
    while $j \leq n$
      $j = 2 * j$
    $i = i - 1$

loop1: $\Theta($ $n \log n$ $)$

loop2: $\Theta($ $\log \log n$ $)$

loop3: $\Theta($ n $)$

⇑⇑⇑ _____ / 6 Punkt(e) ⇑⇑⇑

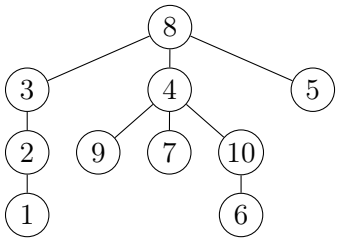
a) Gegeben ist ein ungerichteter Graph als Elternarray.

1	2	3	4	5	6	7	8	9	10
2	3	8	8	8	10	4	0	4	4

Geben Sie eine Adjazenzliste für diesen ungerichteten Graph an.

Lösungsskizze:

Knoten	Nachbarn
1	2
2	1, 3
3	2, 8
4	7, 8, 9, 10
5	8
6	10
7	4
8	3, 4, 5
9	4
10	4, 6



Der Graph ist hier nur für eine bessere Übersicht gedacht und gehört nicht zur Lösung.

⇑⇑⇑ _____ / 8 Punkt(e) ⇑⇑⇑

b) Zeigen oder Widerlegen Sie folgende Aussage:

Für jedes $n \in \mathbb{N}_{>0}$ existiert eine Reihenfolge in der die Zahlen $1, \ldots, n$ in einen binären Suchbaum eingefügt werden können, sodass der resultierende Baum auch ein gültiger Max-Heap ist.

Lösungsskizze: Für $n \geq 3$ existiert keine solche Reihenfolge, da in einem Max-Heap das größte Elemente in der Wurzel ist. Ein Suchbaum in dem das größte Element in der Wurzel ist, hat jedoch einen leeren rechten Teilbaum, d.h. er kann kein fast vollständiger Binärbaum und somit auch kein Heap sein.

⇑⇑⇑ _____ / 10 Punkt(e) ⇑⇑⇑

a) Gegeben ist folgende Hashtabelle der Größe 13. Es wird doppeltes Hashing mit den folgenden Funktionen verwendet:

$$f_i(k) = (h_1(k) + i \cdot h_2(k)) \bmod 13$$
$$h_1(k) = 6k \bmod 13$$
$$h_2(k) = 5 - (k \bmod 5)$$

0	1	2	3	4	5	6	7	8	9	10	11	12
	24	35	5	18	2	14	3		21	1	4	28

Fügen Sie die folgenden Elemente in der gegebenen Reihenfolge in die Hashtabelle ein.

$$1, 2, 3, 4, 5$$

Rechenhilfe:

k	1	2	3	4	5
$h_1(k)$	6	12	5	11	4

⇑⇑⇑ _____ / 10 Punkt(e) ⇑⇑⇑

b) Gegeben ist folgende Hashtabelle der Größe 13. Es wird Hashing mit linearem Austesten mit den Hashfunktionen $h_i(k) = (4k + i) \bmod 13$ verwendet.

0	1	2	3	4	5	6	7	8	9	10	11	12
3	5		30	14	1	4	44	28	25	2	32	29

Fügen Sie die folgenden Elemente in der gegebenen Reihenfolge in die Hashtabelle ein.

$$1, 2, 3, 4, 5$$

Rechenhilfe:

k	1	2	3	4	5
$h_0(k)$	4	8	12	3	7

⇑⇑⇑ _____ / 10 Punkt(e) ⇑⇑⇑

Aufgabe 5: Erdős-Zahl

$8 + 8 + 4 = 20$ Punkt(e)

Paul Erdős war ein ungarischer Mathematiker, der etwa 1 500 Artikel gemeinsam mit über 500 Koautoren[2] publizierte. Ihm zu Ehren wurde die Erdős-Zahl $z(\cdot)$ definiert. Sei $A = \{a_1, \ldots, a_n\}$ die Menge der Autoren. Paul Erdős wird durch $a_1 \in A$ repräsentiert.

- Dann hat Paul Erdős die Zahl $z(a_1) = 0$.

Für jeden anderen Autor $a_i \in A$ mit $i > 1$ gilt:

- a_i hat die Zahl $z(a_i) = k{+}1$ falls k die kleinste Erdős-Zahl unter den Koautoren von a_i ist.
- a_i hat die Zahl $z(a_i) = \infty$, wenn a_i keinen Koautor mit endlicher Erdős-Zahl hat.

Im folgenden wollen wir die Erdős-Zahl von allen Autoren A berechnen. Als **Eingabe** dient ein ungerichteter Graph $G = (A, E)$. Es existiert eine Kante $\{a, b\} \in E$, genau dann wenn Autoren a und b Koautoren sind, d.h. mindestens einen Artikel gemeinsam verfassten. Die **Ausgabe** soll ein Array $Z[1 \ldots n]$ sein, wobei $Z[i]$ die Erdős-Zahl $z(a_i)$ von Autor a_i speichert.

a) Beschreiben Sie **natürlich-sprachlich** einen möglichst effizienten Algorithmus Erdos, der dieses Problem löst. Dieser soll auf einem aus der Vorlesung bekannten Algorithmus VL aufbauen. **Nennen Sie VL und diskutieren Sie die notwendigen Modifikationen** um Z zu berechnen. Begründen Sie kurz die **Korrektheit Ihres Ansatzes** (kein Beweis notwendig).

Hierbei können Sie VL als Blackbox verwenden, das heißt es müssen insbesondere keine Datenstrukturen (o.Ä.) diskutiert werden.

Lösungsskizze: Beobachte, dass es sich bei Erdős-Zahlen um die Kürzeste-Weg-Längen zwischen Paul Erdős und allen anderen Autoren handelt (Äquivalenz kann trivial gezeigt werden, indem man durch zwei Widersprüche etabliert, dass für alle a_i: $z(a_i) \leq \ell$ und $z(a_i) \geq \ell$ gilt, wobei ℓ die Länge eines kürzesten a_1-a_i-Weges ist.)
Da der Graph ungewichtet ist, können wir das zu Grunde liegende Single-Source-Shortest-Path mittels Breitensuche (BFS) lösen: sei $d(v)$ die Tiefe eines Knotens v in einem beliebigen BFS Baum von G in a_1 startend; sei $d(v) = \infty$, wenn v nicht erreicht wird. Dann ergibt sich die Erdős-Zahl von Autor $a \in A$ als $z(a) = d(a)$.
Algorithmus Erdos funktioniert wie folgt: Wir initialisieren zunächst $Z[1] \leftarrow 0$ und alle anderen Einträge mit $Z[2 \ldots n] \leftarrow \infty$. Nun führen BFS aus, wobei wir bei aktivem Knoten a_i für jeden noch unbesuchten Nachbarn a_j den Eintrag $Z[j] \leftarrow Z[i]{+}1$ setzen. Beobachte, dass wir somit auch kein `besucht` Array benötigen und nur testen müssen, ob $Z[j] < \infty$ endlich ist.

[2] Zwei Autoren sind genau dann Koautoren, wenn sie mindestens ein Papier gemeinsam publizierten.

b) Beschreiben Sie Ihren effizienten Algorithmus in **Pseudo-Code**. Definieren und initialisieren Sie zunächst alle genutzten Datenstrukturen (inkl. der Graph-Repräsentation).

Lösungsskizze: Wir speichern G als Adjazenzliste. Dies erlaubt uns die Nachbarschaft $N[v]$ eines Knotens v in Zeit $|N[v]|$ zu enumerieren.

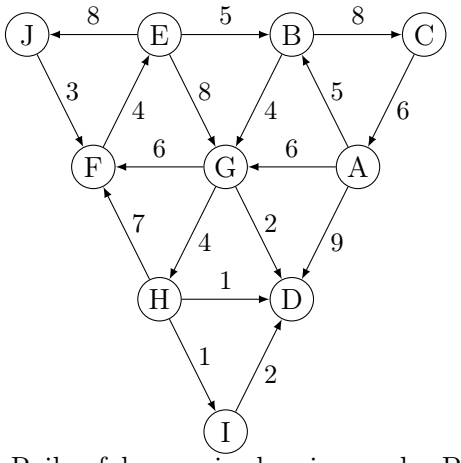
- Allokiere Array $Z[1 \dots n]$ und setze $Z[1] \leftarrow 0$ und $Z[2 \dots n] \leftarrow \infty$
- Sei $Q \leftarrow [1]$ eine Queue von Knoten-Ids, die initial nur a_1 enthält
- Solange Q nicht leer ist wiederhole:
  - Speichere das erste Element der Queue Q in i und entferne es aus Q
  - Für jeden Nachbarn a_j in $N[a_i]$:
    Wenn $Z[j] \neq \infty$: überspring j (denn a_j wurde bereits besucht)
    Setze $Z[j] \leftarrow Z[i] + 1$
    Füge j in Q ein
- Gebe Z zurück

(Alternativ könnten etwa auch in Queue Tupel der Form $(i, z(a_i))$ gepusht werden)

c) Analysieren Sie die Laufzeit Ihres Algorithmus **asymptotisch exakt** für den Fall, das alle Autoren eine endliche Erdős-Zahl haben.

Lösungsskizze: Unter der Annahme, dass alle Autoren eine endliche Erdős-Zahl haben, ist G zusammenhängend. Da wir einen Label-Setting-Algorithmus haben (die Werte von $Z[i]$ werden nach der Initialisierung höchstens einmal gesetzt), landet dann jeder Knoten exakt einmal in der Queue. Somit wird in der For-Schleife jede Kante exakt zwei Mal betrachtet. Wir führen also $\Theta(|A|+|E|)$ Operationen aus; da G zusammenhängend ist, gilt $|A| = \mathcal{O}(|E|)$ und somit hat ERDOS in diesem Fall eine Laufzeit von $\Theta(|E|)$.

a) Wenden Sie Dijkstras Algorithmus auf folgenden gewichteten, gerichteten Graph mit **Startknoten** A an.



Geben Sie die Knoten in der Reihenfolge an, in der sie aus der Prioritätswarteschlange entfernt werden.

Reihenfolge: A, *B, G, D, H, I, F, C, E, J*

Geben Sie den ersten Knoten an für den die Distanz von einem Wert kleiner ∞ erneut reduziert wird, bevor er aus der Prioritätswarteschlange entfernt wird.

Knoten: *D*

⇑⇑⇑ _____ / 12 Punkt(e) ⇑⇑⇑

b) Sei $G = (V, E)$ ein gewichteter, gerichteter Graph und $w : E \to \mathbb{R}_{\geq 1}$ eine Kantengewichtsfunktion. Wir betrachten die folgende Transformation der Kantengewichte:

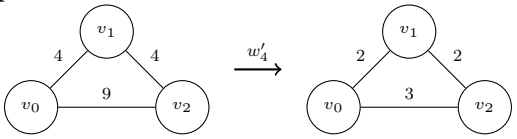
$$w'_a(e) = 2^{\log_a(w(e))}$$

i) **(4 Punkte)** Geben Sie ein $a \in \mathbb{N}_{>1}$ an, sodass die kürzesten Wege bzgl. w unter der Transformation w'_a erhalten bleiben und begründen Sie warum.

Lösungsskizze: Für $a = 2$ gilt also $w'_2(e) = w(e)$, d.h. für $a = 2$ bleiben alle kürzesten Wege erhalten, da sich die Gewichte nicht ändern.

ii) **(4 Punkte)** Geben Sie ein Beispiel für ein $a \in \mathbb{N}_{>1}$ an, sodass die kürzesten Wege bzgl. w unter der Transformation w'_a *nicht* erhalten bleiben und begründen Sie an einem Beispiel warum.

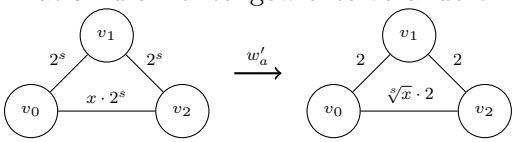
Lösungsskizze:
Für $a = 4$ gilt $w'_4(e) = \sqrt{w(e)}$. Dass die kürzesten Wege nicht erhalten bleiben zeigt folgendes Beispiel:



Für w ist der Weg von v_0 über v_1 zu v_2 der Kürzeste. Für w'_4 ist es der direkte Weg v_0, v_2 ohne Umweg über v_1.

iii) **(4 Punkte)** Verallgemeinern Sie das Beispiel aus ii) auf alle a außer das, das in Aufgabenteil i) verwendete. Beachten Sie, dass das in Aufgabenteil i) verwendete a das Einzige ist für das die kürzesten Wege erhalten bleiben.

Lösungsskizze: Die Transformation lässt sich mit den Logarithmengesetzen umformen zu: $w'_a(e) = w(e)^{1/\log_2(a)}$, da $\log_a w(e) = \frac{\log_2(w(e))}{\log_2(a)}$ gilt.
Für alle $a > 2$ sei $s = \log_2(a)$, d.h es gilt $s > 1$. Wir betrachten folgenden Graph und wie die Transformation die Kantengewichte verändert.



Sei für w der Pfad (v_0, v_1, v_2) der kürzeste Pfad von v_0 nach v_2 und für w'_a sei es der Pfad (v_0, v_2). Dann gilt sowohl

$$2 \cdot 2^s < x \cdot 2^s \qquad \Rightarrow \qquad 2 < x$$

als auch

$$2 \cdot 2 > \sqrt[s]{x} \cdot 2 \qquad \Rightarrow \qquad 2^s > x$$

Für $s > 1$ existieren x die beide Ungleichungen erfüllen, z.B. $x = 2^{1+(s-1)/2}$ und damit bleiben kürzeste Wege unter w'_a für $a > 2$ nicht erhalten.

⇑⇑⇑ _____ / 12 Punkt(e) ⇑⇑⇑

Aufgabe 7: Huffman-Codes — 12 Punkt(e)

Geben Sie zu jedem Zeichen das Codewort eines Huffman-Codes an der zu folgender Häufigkeitstabelle gehört.

a	b	c	d	e	f	g
3	3	1	1	7	2	4

Immer wenn zwei Teilbäume vereint werden, können Sie beliebig anordnen. Sollten Sie zwischen zwei gleichwertigen Knoten auswählen müssen, können Sie beliebig entscheiden.

Der Huffman-Baum muss nicht angegeben werden und wird auch nicht bewertet.

a ≙ 100

b ≙ 101

c ≙ 0100

d ≙ 0101

e ≙ 11

f ≙ 011

g ≙ 00

⇑⇑⇑ _____ / 12 Punkt(e) ⇑⇑⇑

Aufgabe 8: Augensumme mehrerer Würfel — $4+6+8+2+4 = 24$ Punkt(e)

Gegeben seien n Spielwürfel mit Ziffern 1 bis 6. Wir werfen diese nacheinander, und notieren die Augenzahlen $(z_1, \ldots, z_n)$ mit $z_i \in \{1,2,3,4,5,6\}$ für alle $1 \leq i \leq n$. Gesucht ist die Anzahl $N[n,k]$ an Kombinationen $(z_1, \ldots, z_n)$, so dass die Augensumme $\sum_{i=1}^{n} z_i$ exakt den Wert k mit $n \leq k \leq 6n$ annimmt, d.h.

$$N[n,k] \quad = \quad \left| \left\{ (z_1, \ldots, z_n) \;\middle|\; z_i \in \{1,2,3,4,5,6\}\ \forall 1 \leq i \leq n \text{ mit } \sum_{i=1}^{n} z_i = k \right\} \right|.$$

Hinweis: Sie können annehmen, dass die Eingabe $n \geq 1$ und $n \leq k \leq 6n$ erfüllt.

a) Betrachten Sie folgenden naiven Algorithmus: wir testen *jede* Kombination $(z_1, \ldots, z_n)$ mit $1 \leq z_i \leq 6$ für alle $1 \leq i \leq n$, und zählen wie viele Kombination die Augensumme k ergeben. Geben Sie eine möglichst scharfe untere Schranken für die Laufzeit dieses Algorithmus an. Begründen Sie Ihre Antwort kurz.

Lösungsskizze: Es existieren 6^n Kombinationen, die getestet werden müssen. Hierbei muss mindestens konstante Zeit pro Test aufgewendet werden. Somit ergibt sich eine untere Schranke von $\Omega(6^n)$ (durch geschickte Enumeration ist diese Schranke exakt).

⇑⇑⇑ _____ / 4 Punkt(e) ⇑⇑⇑

b) Wir möchten das Problem nun mit Hilfe eines **dynamischen Programms** lösen. Hierfür berechnen wir $N[n,k]$ **rekursiv**. Geben Sie eine geeignete Rekursionsgleichung inklusive minimaler notwendiger Basisfälle an.

Rekursion: $N[n,k] =$ $\sum_{i=1}^{\min(k-1,6)} N[n-1, k-i]$

Basisfälle:

Lösungsskizze:

$$N[1,k] = \begin{cases} 1 & \text{wenn } 1 \leq k \leq 6 \\ 0 & \text{sonst} \end{cases}$$

Sollte die Rekursionsgleichung immer über $i = 1 \ldots 6$ summieren, müssen noch Basisfälle $N[n,k] = 0$ für $-5 \leq k \leq 0$ angegeben werden.

⇑⇑⇑ _____ / 6 Punkt(e) ⇑⇑⇑

c) Beschreiben Sie einen Algorithmus in **Pseudo-Code**, der Ihre Rekursionsgleichung $N[n,k]$ mittels einer $n \times k$ Matrix effizient löst. Nutzen Sie `return` um das Endergebnis zurückzugeben.

Lösungsskizze:
- Allokiere $N[1..n, 1..k]$ als $n \times k$ Matrix
- For j in $1 \dots k$:
  falls $j \leq 6$ setze $N[1,j] \leftarrow 1$ sonst setze $N[1,j] \leftarrow 0$
- For i in $2 \dots n$:
  For j in $1 \dots k$:
  $N[i,j] \leftarrow \sum_{\ell=1}^{\min(j-1,6)} N[i-1, j-\ell]$
- Return $N[n,k]$

`return` $N[n,k]$

⇑⇑⇑ _____ / 8 Punkt(e) ⇑⇑⇑

d) Leiten Sie die Laufzeit Ihres Algorithmus asymptotisch exakt her.

Lösungsskizze: Es gibt $n \cdot k$ Einträge in der Matrix; pro Eintrag fallen konstante Initialisierungs- oder Berechnungskosten an. Die Gesamtlaufzeit beträgt somit $\Theta(n \cdot k)$.

⇑⇑⇑ _____ / 2 Punkt(e) ⇑⇑⇑

e) Beschreiben Sie kurz natürlich-sprachlich, wie sich der Speicherverbrauch von $\Theta(n \cdot k)$ auf $\mathcal{O}(n)$ reduzieren lässt.

Lösungsskizze: In der Rekursion wird nur die jeweils vorherige Zeilen ($n \leftarrow n-1$) benötigt. Somite müssen nur die jeweils vorherige und die aktuelle Zeile im Speicher gehalten werden; die zwei Arrays können mit jeder neuen Zeile ihre Rolle wechseln. Jede Zeile besteht aus $k = \mathcal{O}(n)$ Einträgen.

⇑⇑⇑ _____ / 4 Punkt(e) ⇑⇑⇑

Wichtig: Lösungen auf dieser Seite werden nur dann berücksichtigt, wenn bei der entsprechenden Aufgabe ein Hinweis auf Seite 15 platziert wurde.

2011000100018